DATA SCIENCE RETREAT

SQL

A crash course

Agenda

- Intro to Relational Databases
- SQL Basics
- Joins, Aggregations, Subqueries, Window/Analytic Functions
- Tricks and Tips

The RDBMS Landscape

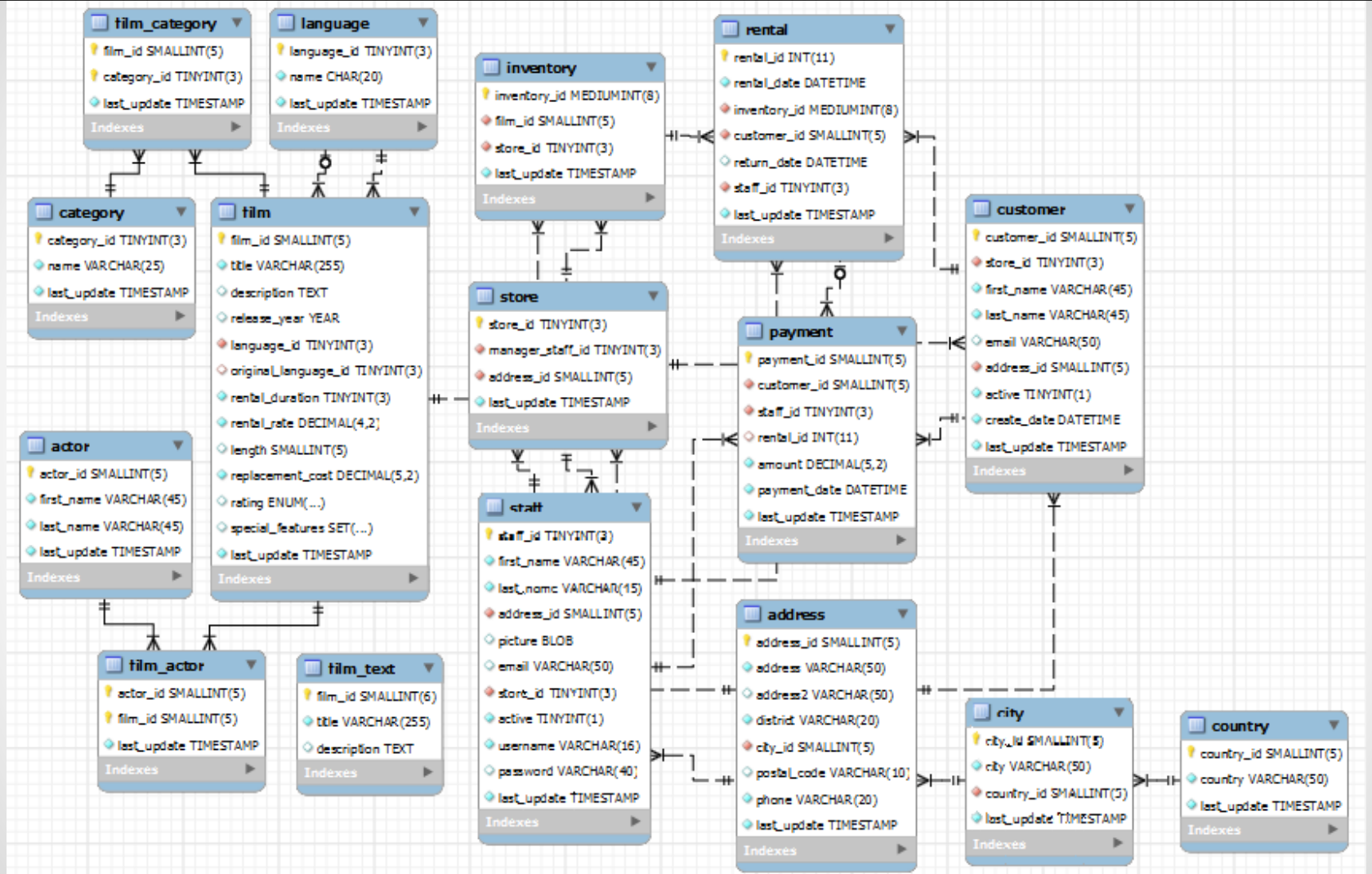
- How did we get here
  - Tapes
  - Sequential files
  - Indexed files
  - Hierarchical Databases
  - Relational Databases
- Major relational DBs
  - SQLite
  - Oracle
  - SQL Server
  - IBM DB2
  - MySQL/MariaDB
  - PostgreSQL
  - Amazon RDS
  - Amazon Redshift
  - SAP Hana
  - Hive
  - SparkSQL

Basic Concepts

- CAP Theorem: pick any two
  - Consistency
  - Availability
  - Partition tolerance
- ACID:
  - Atomicity: indivisible (all or nothing) transactions
  - Consistency: you always get to either before or after the transaction
  - Isolation: each transaction sees things as they were before until it is finished
  - Durability: if your transaction finished, (and the DB is ok), it is safe.

Checklist

- Install pgAdmin from pgadmin.org
- Connect to
  - server: 192.168.2.148
  - port: 5432
  - database: dvdrental
  - user: dsr
  - password: dsr
  - `select count(*) from film` -- should return 1000
  - don't modify anything — no INSERT, UPDATE, DELETE, or ALTER commands


film_category
film_id SMALLINT(5)
category_id TINYINT(3)
last_update TIMESTAMP
Indexes
language
language_id TINYINT(3)
name CHAR(20)
last_update TIMESTAMP
Indexes
inventory
inventory_id MEDIUMINT(8)
film_id SMALLINT(5)
store_id TINYINT(3)
last_update TIMESTAMP
Indexes
rental
rental_id INT(11)
rental_date DATETIME
inventory_id MEDIUMINT(8)
customer_id SMALLINT(5)
return_date DATETIME
staff_id TINYINT(3)
last_update TIMESTAMP
Indexes
category
category_id TINYINT(3)
name VARCHAR(25)
last_update TIMESTAMP
Indexes
film
film_id SMALLINT(5)
title VARCHAR(255)
description TEXT
release_year YEAR
language_id TINYINT(3)
original_language_id TINYINT(3)
rental_duration TINYINT(3)
rental_rate DECIMAL(4,2)
length SMALLINT(5)
replacement_cost DECIMAL(5,2)
rating ENUM(...)
special_features SET(...)
last_update TIMESTAMP
Indexes
store
store_id TINYINT(3)
manager_staff_id TINYINT(3)
address_id SMALLINT(5)
last_update TIMESTAMP
Indexes
payment
payment_id SMALLINT(5)
customer_id SMALLINT(5)
staff_id TINYINT(3)
rental_id INT(11)
amount DECIMAL(5,2)
payment_date DATETIME
last_update TIMESTAMP
Indexes
customer
customer_id SMALLINT(5)
store_id TINYINT(3)
first_name VARCHAR(45)
last_name VARCHAR(45)
email VARCHAR(50)
address_id SMALLINT(5)
active TINYINT(1)
create_date DATETIME
last_update TIMESTAMP
Indexes
actor
actor_id SMALLINT(5)
first_name VARCHAR(45)
last_name VARCHAR(45)
last_update TIMESTAMP
Indexes
staff
staff_id TINYINT(3)
first_name VARCHAR(45)
last_name VARCHAR(45)
address_id SMALLINT(5)
picture BLOB
email VARCHAR(50)
store_id TINYINT(3)
active TINYINT(1)
username VARCHAR(16)
password VARCHAR(40)
last_update TIMESTAMP
Indexes
address
address_id SMALLINT(5)
address VARCHAR(50)
address2 VARCHAR(50)
district VARCHAR(20)
city_id SMALLINT(5)
postal_code VARCHAR(10)
phone VARCHAR(20)
last_update TIMESTAMP
Indexes
city
city_id SMALLINT(5)
city VARCHAR(50)
country_id SMALLINT(5)
last_update TIMESTAMP
Indexes
country
country_id SMALLINT(5)
country VARCHAR(50)
last_update TIMESTAMP
Indexes
film_actor
actor_id SMALLINT(5)
film_id SMALLINT(5)
last_update TIMESTAMP
Indexes
film_text
film_id SMALLINT(6)
title VARCHAR(255)
description TEXT
Indexes

Database Structures

- Storage (HDs, SSDs, memory, etc.)
- Files (sometimes memory-mapped)
  - Cache is a whole different story
- Tablespace
  - Logical allocation of space in files for database objects
- Database
  - Essentially a larger collection of related data (i.e., for an application)
- Schemas
  - Logical organization of tables by user or subject within a database
- Database Objects
  - Tables, Views, indexes, statistics, functions, triggers, etc.
- Security
  - Users, roles, privileges

Relational Operations

- Basic
  - Selection: subsetting
  - Union, Difference and Intersection
  - Projection: selecting fields
- Row Processing
  - Aggregation
  - Hierarchy/Recursion
- Joins
  - Natural Join
  - Cartesian Product
  - Equijoin: conditional (never mind)
  - Semijoin and antijoin: filtering
  - Division (never seen it)
  - Outer joins
    - Left
    - Right
    - Full

Data Types

- NULL
- Numbers (link is for Postgres only)
  - Integers and floating point (same as in languages)
  - NUMERIC (arbitrary precision, decimal exact and SLOW!!!)
  - MONEY
- Text
  - CHAR(n), VARCHAR(n), TEXT (check you database for encoding support and configuration)
  - Bytea (raw byte strings, can store blobs – but not really large objects)
- Date/Time
  - DATE, TIME, Timestamp, Interval
- Boolean
  - TRUE/FALSE
- Enumerated
  - Similar to ordered factors, have to be defined by CREATE TYPE xxx AS ENUM
  - Careful, converting to integer is not easy
- Text Search
  - More on that later
- Others
  - Geometric, network addresses, bit strings, UUID, XML, JSON, arrays, composite

SQL Statements: DDL

- DDL Creates, changes or deletes database objects
  - CREATE
  - ALTER
  - DROP
  - Clauses, options and parameters depend on the object type
- Example: CREATE TABLE
  - ```
    create table my_table (
        name varchar(30)
                        CONSTRAINT PK_MY_TABLE PRIMARY KEY
                                    USING INDEX TABLESPACE pg_default
                                    deferrable,
        age numeric(3) DEFAULT NULL
                        CONSTRAINT AGE_RANGE CHECK (age between 0 and 130)
    )
        TABLESPACE pg_default
    ```
- Important: the default for DDL is to execute immediately, no rollback or undo.
    ```

SQL Statements: DML

- DML manipulates data inside database objects (mostly tables and views, but can also call functions)
- Some DML may activate side actions (such as triggers)
- DML Statements
  - SELECT … FROM table(s)…
  - SELECT INTO table FROM …
  - INSERT INTO table(columns) VALUES
  - INSERT INTO table AS SELECT
  - UPDATE table SET column = value WHERE…
  - DELETE FROM table WHERE…

Examples using CREATE, INSERT, UPDATE, ALTER

- Create a table named "SILLY" with First Name, Last Name and Date of Birth
  - CREATE TABLE silly (first varchar(20), last varchar(20), dob date);
- Insert data into that table
  - INSERT INTO silly (first, last, dob) VALUES ('Fred', 'Flintstone', '1960-09-30');
- Create a table called "SILLIER" using SELECT INTO from SILLY
  - SELECT * INTO sillier FROM silly;
- Change the release year for films to be randomly spread between 2001 and 2011
  - UPDATE film SET release_year = 2001 + cast(random() * 10 as integer);
- Add a birth date to the ACTOR table
  - ALTER TABLE actor ADD COLUMN birthdate date;

SQL Query

```
SELECT  <attributes>
FROM    <one or more relations>
WHERE   <conditions>
```

Inside the SELECT Statement

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]

where from_item can be one of:

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] | column_definition [, ...] ) ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]

and with_query is:

    with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert | update | delete )

TABLE [ ONLY ] table_name [ * ]
```

Conceptual Evaluation Strategy

- Semantics of an SQL query are defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of relation-list.
  - Discard resulting tuples if they fail qualifications.
  - Delete attributes that are not in target-list.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.

The WHERE Clause

- This is where you filter your rows.
- All operators and functions we've seen can be used here
  - =, >, <, AND, OR, NOT, IS, etc
  - IN, EXISTS, LIKE, ~
- You can do inner joins here (A.COLUMN = B.COLUMN)

Tricky question

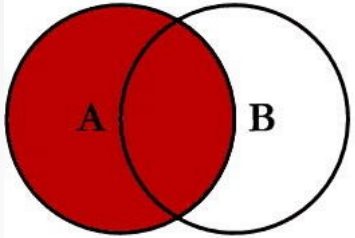
```
SELECT *
FROM   Person
WHERE  age < 25  OR  age >= 25
```

Can it be that some Persons are not included?

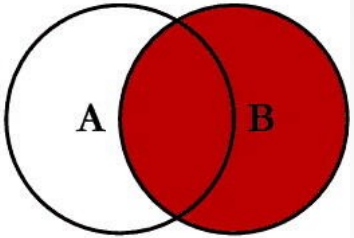
WARNING

- SQL has a 3-state boolean logic system. WHERE only accepts rows where the condition is TRUE (i.e. not FALSE, but also not UNKNOWN).

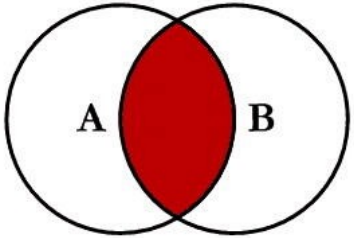
SQL JOINS



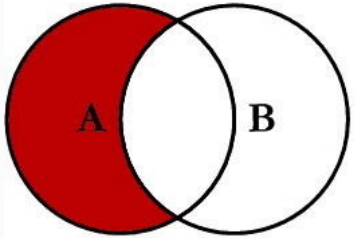

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



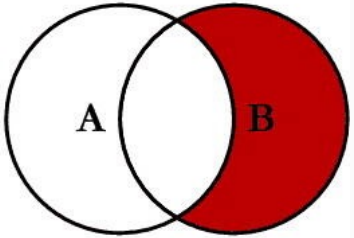

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



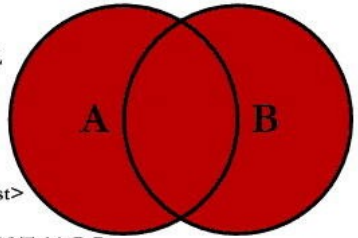

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



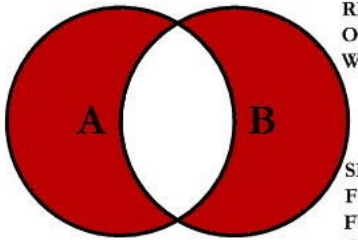

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Joins

- Inner Joins
  - SELECT ... FROM A, B WHERE A.COLUMN_A = B.COLUMN_B
  - SELECT ... FROM A INNER JOIN B ON A.COLUMN_A = B.COLUMN_B
  - If column names are the same, ON becomes USING COLUMN_NAME
- Outer left
  - SELECT ... FROM A LEFT JOIN B ON A.COLUMN_A = B.COLUMN_B
  - All rows from A, NULLs for B columns when no match
- Outer Right
  - SELECT ... FROM A RIGHT JOIN B ON A.COLUMN_A = B.COLUMN_B
  - Actually, just the same as the left, if you change A and B
- Full Outer
  - SELECT ... FROM A FULL JOIN B ON A.COLUMN_A = B.COLUMN_B
  - All rows from both, NULLs for the side that does not match
  - You can find all that don’t match by filtering on NULL for the key columns of A or B

Join Exercises

- List all film titles with their actors' names
- Titles of films that are not in the inventory
- The titles of all films returned on 2005-05-27

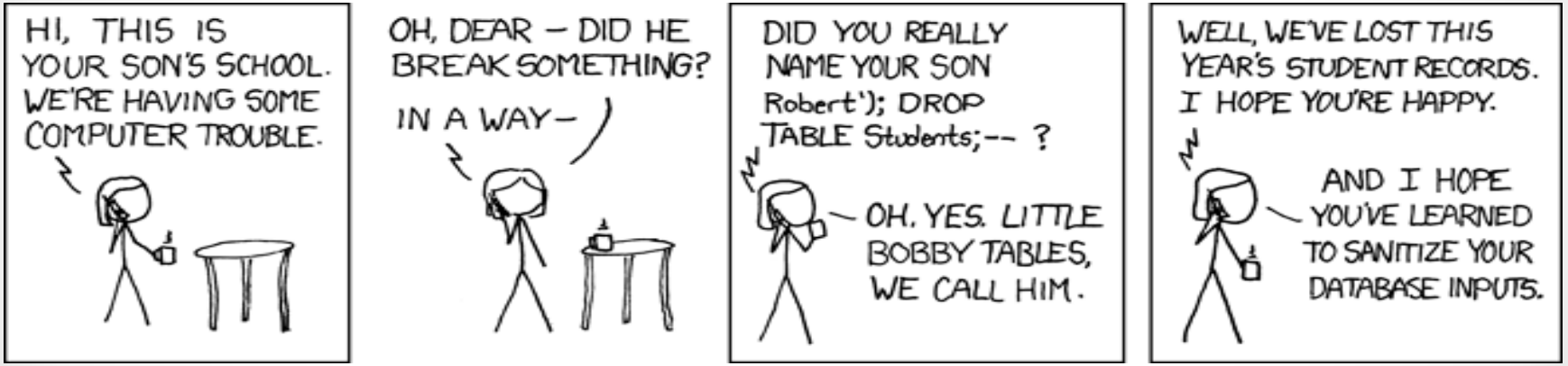
films that are not in the inventory

```
SELECT
        film.film_id,
        film.title,
        inventory_id
FROM
        film
LEFT JOIN
        inventory ON inventory.film_id = film.film_id
WHERE
        inventory.film_id IS NULL;
```

all films returned on 2005-05-27

```
SELECT
  film.title
FROM
  film
  JOIN inventory ON inventory.film_id = film.film_id
  JOIN rental ON
    rental.inventory_id = inventory.inventory_id AND
    CAST (return_date AS DATE) = '2005-05-27';
```

Little Bobby Tables revisited



Aggregations

- GROUP BY
- HAVING
  - Similar to WHERE, but conditions are over aggregates

```
select C.NAME, sum(p.amount)
        FROM CATEGORY C
                JOIN FILM_CATEGORY FC USING (CATEGORY_ID)
                JOIN FILM F USING (FILM_ID)
                JOIN INVENTORY I USING (FILM_ID)
                JOIN RENTAL R USING (INVENTORY_ID)
                JOIN PAYMENT P USING (RENTAL_ID)
        GROUP BY NAME
        HAVING SUM(P.AMOUNT) > 4000
        ORDER BY SUM(P.AMOUNT) DESC
        LIMIT 10
```

- OVER (PARTITION BY)

Other elements

- SELECT DISTINCT
- UNION, UNION ALL, MINUS, INTERSECT
- Naming columns: The “AS” clause

Aggregation Exercises

- customers ranked by how much they've spent
- customers who have spent more than 200
- stores with more than 200 customers
- the number of rentals from each category

```
-- customers by customer_id ranked by how much they've spent
SELECT      customer_id, SUM (amount)
FROM        payment
GROUP BY    customer_id
ORDER BY    SUM (amount) DESC;

WITH customer_totals AS (select customer_id, SUM (amount) as total FROM payment group by customer_id)
SELECT      customer_id, total
FROM        customer_totals
ORDER BY    total DESC;
```

```
-- customers who have spent more than 200
SELECT      customer_id, SUM (amount)
FROM        payment
GROUP BY    customer_id
HAVING      SUM (amount) > 200;

-- stores with more than 100 customers
SELECT      store_id, COUNT (customer_id)
FROM        customer
GROUP BY    store_id
HAVING      COUNT (customer_id) > 100;
```

number of rentals from each category

```
SELECT count(*), category.name from rental
JOIN inventory on rental.inventory_id = inventory.inventory_id
JOIN film_category on inventory.film_id = film_category.film_id
JOIN film on inventory.film_id = film.film_id
JOIN category on film_category.category_id = category.category_id
GROUP BY category.name
ORDER BY count desc
```

Subqueries

```
SELECT SUM (Sales) FROM Store_Information
WHERE Store_Name IN
  (SELECT Store_Name FROM Geography
   WHERE Region_Name = ‘West');
```

Correlated subquery

```
SELECT employee_number, name
  FROM employees AS emp
  WHERE salary > (
    SELECT AVG(salary)
      FROM employees
      WHERE department = emp.department
  );
```

Optimizing correlated subqueries

```
-- This subquery is not correlated with the outer query, and is therefore
-- executed only once, regardless of the number of employees.
SELECT employees.employee_number, employees.name
  FROM employees
    INNER JOIN
      (SELECT department, AVG(salary) AS department_average
        FROM employees
        GROUP BY department) AS temp
    ON employees.department = temp.department
  WHERE employees.salary > temp.department_average;
```

Subqueries

- IN
  - (column_list) IN (list)
  - Can be a list of values
  - Can be another select
  - You can actually check for tuples, like (first_name, last_name) in (select first_name, last_name from...)
- EXISTS
  - Will be true if at least one comparison satisfies the condition
  - Good for checking if something is on a list (correlated subquery)
- ANY
  - Can check for more than IN. Any comparison operator goes
  - IN is the same as =ANY
- ALL
  - Same as ANY, but will be true if the condition holds for all cases.
  - NOT IN is the same as <> ALL

Subquery Exercises

- names of all customers who returned a rental on 2005-05-27
- films whose rental_rate is higher than the average rental_rate
- names of customers who have made a payment
  - with a subquery
  - with a JOIN

names of customers who returned a rental on 2005-05-27

```
SELECT
        first_name, last_name
FROM
        customer
WHERE
        customer_id IN (
                SELECT customer_id
                FROM rental
                WHERE CAST (return_date AS DATE) = '2005-05-27'
        );
```

films whose rental rate is higher than the average rental rate

```
SELECT
    film_id, title, rental_rate
FROM
    film
WHERE
    rental_rate > (SELECT AVG (rental_rate) FROM film);
```

names of customers who have made a payment

```
SELECT
    first_name, last_name
FROM
    customer
WHERE
    EXISTS (
        SELECT
            1
        FROM
            payment
        WHERE
            payment.customer_id = customer.customer_id
    );
```

what about using a JOIN?

```
SELECT
  DISTINCT
    first_name, last_name
FROM
    customer
JOIN
  payment ON
    payment.customer_id =
      customer.customer_id
```

Window Functions

- background: https://robots.thoughtbot.com/postgres-window-functions
- exercise: find the most frequently rented film from each category

number_of_rentals	category	title	rank_in_category
34	Travel	Bucket Brotherhood	1
33	Foreign	Rocketeer Mother	1
32	Animation	Juggler Hardly	1
32	Games	Forward Temple	1
32	Games	Grit Clockwork	1
32	Music	Scalawag Duck	1
32	New	Ridgemont Submarine	1
31	Children	Robbers Joon	1
31	Classics	Timberland Sky	1
31	Comedy	Zorro Ark	1
31	Documentary	Wife Turn	1
31	Drama	Hobbit Alien	1
31	Family	Apache Divine	1
31	Family	Network Peak	1
31	Family	Rush Goodfellas	1
31	Sci-Fi	Goodfellas Salute	1
30	Action	Suspects Quills	1
30	Action	Rugrats Shakespeare	1
30	Horror	Pulp Beverly	1
29	Sports	Gleaming Jawbreaker	1
29	Sports	Talented Homicide	1
30	Animation	Dogma Family	2
30	Children	Idols Snatchers	2

the most rented film in each category

```
SELECT count(*) as number_of_rentals, category.name as category,
film.title, dense_rank()
  OVER (PARTITION BY category.name ORDER BY count(*) desc)
  AS rank_in_category
FROM rental
JOIN inventory ON rental.inventory_id = inventory.inventory_id
JOIN film_category ON inventory.film_id = film_category.film_id
JOIN film ON inventory.film_id = film.film_id
JOIN category ON film_category.category_id = category.category_id
GROUP BY category.name, film.film_id
ORDER BY rank_in_category asc, number_of_rentals desc, category asc;
```

latest trends

- SQL for distributed big data frameworks
- Apache Calcite
- Streaming SQL

Optimization

- Joins
- Sorts
- Explain plan
- The Query Optimizer
- When do indexes matter
- Remember about table statistics
- Hints
- Temporary tables

Some other tricks

- Pivoting and reshaping
  - mySQL Example (works on most databases)
- Generating sequences on the fly
  - Use generate_series() to create a list of dates as a subquery, then outer join to your data and you get evenly distributed observations from sparse actual cases
    - Of course, you have to impute missing values
- Sampling
  - SELECT ... ORDER BY random() LIMIT sample_size
- Text processing

The WITH Clause

- Defines other queries which can be used as if views from the main clause
  - WITH RENTALS AS (
    SELECT C.FIRST_NAME, C.LAST_NAME, R.RENTAL_ID, I.FILM_ID
      FROM CUSTOMER C, RENTAL R, INVENTORY I
      WHERE C.CUSTOMER_ID = R.CUSTOMER_ID AND
        R.INVENTORY_ID = I.INVENTORY_ID)
    SELECT F.TITLE, RENTALS.FIRST_NAME, RENTALS.LAST_NAME
      FROM FILM F, RENTALS
      WHERE F.FILM_ID = RENTALS.FILM_ID
      ORDER BY TITLE
- Recursive
  - PostgreSQL allows for a special RECURSIVE syntax which can be used for navigating through a hierarchy
  - Let's try an exercise:
    - Add a SEQUEL_TO column to the FILM table
    - Make 100 movies sequel to another different 100 movies (loops ok, sequels to self no).
    - You can use LIMIT clause to restrict to only 100
    - You can order by random() to take a random sample
  - Now, write a query that returns ALL the sequels to one movie

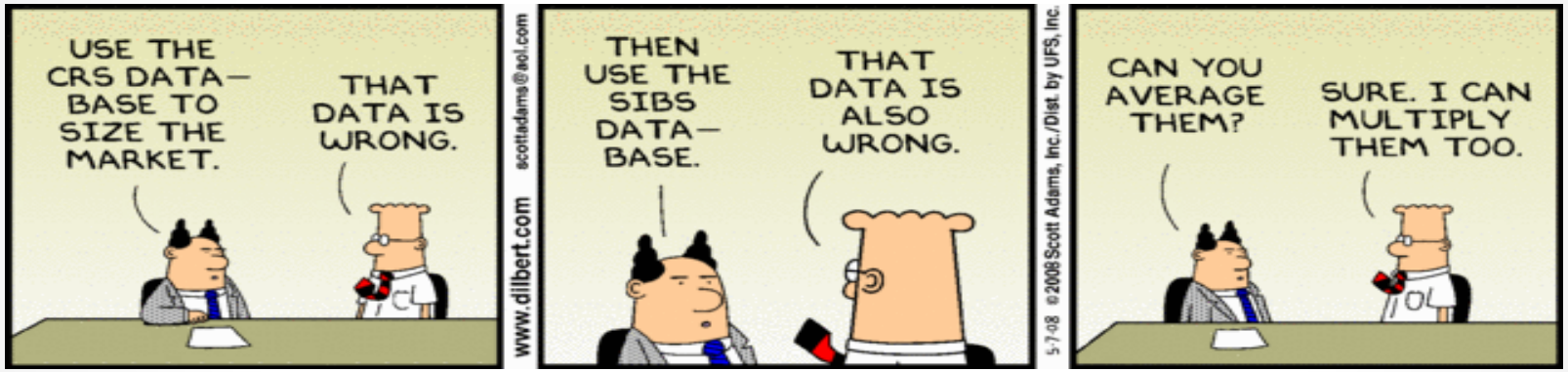
Connecting to Python

```
import psycopg2
import psycopg2.extras

def ResultIter(cursor, arraysize=1000):
    'An iterator that uses fetchmany to keep memory usage down'
    while True:
        results = cursor.fetchmany(arraysize)
        if not results:
            break
        for result in results:
            yield result

conn = psycopg2.connect("dbname=dvdrental user=postgres host=192.168.111.128")
cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
cur.execute("select * from film")
for result in ResultIter(cur):
    print(result)
```

So, now you can do this...



Example of Snowflake Schema